Architecture of the code OPERA (EOLSR Strategic Mode + OSERENA) (document version 3)

Cédric Adjih, Ichrak Amdouni
Inria Paris-Rocquencourt – Team Hipercom2

Contents

1 Introduction

In this document, we present an overview of the code for sensor networks developped by Inria. In the code, the generic prefix "HipSens" is used (which is a concatenation of the two words "HIPercom" and "SENSor"), for instance for file names.

It includes the following modules:

- An implementation of "EOLSR" for routing in strategic mode with the following sub-modules:
  - EOND: EOLSR Neighbor Discovery.
  - EOSTC: EOLSR Strategic Tree Construction. It is subdivised itself in two parts:
    - Base routing tree construction
    - Extensions for the use of the tree for coloring: stability/unstability detection, descendant counting.
- OSERENA: an implementation of an Optimized version of SERENA, for node coloring (in the code, the name "SERENA" is used to designate OSERENA).
- An implementation of OPERA, a general module which is orchestrating the different parts (in `hipsens-opera.c`).
  By abuse of language, the name OPERA is used for the whole stack (EOND + EOSTC + OSERENA + orchestrating module).

1.1 Relation with OLSR

By itself, EOLSR stands for "Energy-efficient Optimized Link State Routing". Here, only the core functionalities have been included, to permit:

- Energy-efficient (or energy-aware) routing. For instance, if a subset of nodes is on external power supply, routes will go through them preferentially (in EOSTC).
- Limited proactive route maintenance: routes are maintained to only one or a few nodes (strategic nodes) by construction of routing trees (in EOSTC). Proactives routes to other nodes are not available.
- Coloring is implemented through an optimized version of OSERENA (which does not requires maintaining nor transmitting the whole 3-hop neighborhood in messages).

- When the sink is not the CPAN, an alternate EOSTC tree can be colored, hence the routing tree construction in EOSTC is complemented by features which actually serve a triple purpose:
  - Being able to have a sink in a separate node from the CPAN.
  - Providing an energy-aware tree to collect data to the sink, in a optimized schedule following the tree structure (following optimization idea from LIMOS).
  - Being able to detect stability and unstability of the tree, along with scheduling coloring restart.

2 General Design

The design goals of the code are the following:

- Sensors: Make it fully functionnal on sensor nodes (with a few KiB of RAM and ROM), i.e. OCARI target nodes.
- Portability: The code must run both on actual sensors and on simulators.

Because none of the main modules in OPERA actually requires real-time or concurrent processing, the goals are reasonnably compatible.

As we have previously indicated, the code includes 3 protocols:

- EOND, the neighbor discovery (from EOLSR strategic mode): this module discovers, and maintains neighbors, through the use of HELLO messages
- EOSTC, the strategic tree construction: this module builds trees to sinks (i.e. nodes which are the root of STC trees), through the use of STC messages. In addition, some trees[1] are designed to be colored through with the protocol OSERENA, and for this reason have additional processing and features to the core tree construction of STCs.
- OSERENA, the node coloring protocol ; this module colors nodes through the use of "Color" messages. It also propagates state of the coloring through "Max Color" messages to the sink.

[1] at most one, normally

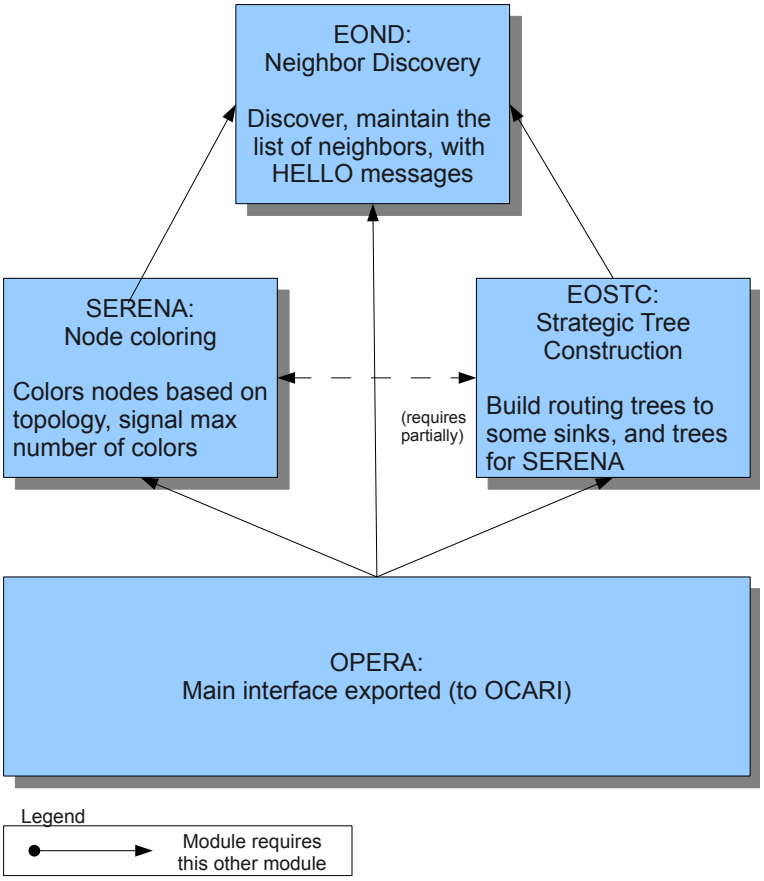


Figure 1: Dependencies between the modules

The modules are designed to be used directly (and they are: for some simulations EOND + OSERENA are instantiated directly) ; however because the whole set of protocols is currently work in progress, an additional module exists:

- OPERA interface, a module designed to encapsulated the three protocols EOND, EOSTC, OSERENA and their interactions. When used the 3 modules of the protocols do not need to be instantiated. All the interaction is done through functions of OPERA, and the functions of the modules EOND, EOSTC, OSERENA are considered to be internal to OPERA.

The dependencies between the modules are illustrated in figure 1
They are described in the next sections.

2.1 Sub-modules EOND, EOSTC and OSERENA

The design of each of the three modules is patterned after the following structure:

- The whole state of each module is stored in a separate C structure[2]. These are `eond_state_t`, `eostc_state_t`, and `serena_state_t`
- The module follows internally an event-based approach, with 3 main functions:
  - A "packet processing" function (to handle the event when one packet has been received).
  - A "packet generation" function (to handle the event when time has come to generate a packet).
  - A "next wake-up time" function (indicating when the module wants to be called again).

This structure transfers most of the responsibility to the main module of OPERA, responsible of orchestrating the 3 sub-modules. For instance, none of the modules considers concurrency, the orchestrating module is in charge of it. [3]

In the same way, the orchestrating module is in charge of arranging scheduling/timers so that each module gets called as indicated by the "next wake-up time" of each function.

2.1.1 Cycle-based API of OPERA

This section documents the lastest API implemented by OPERA with respect to the interaction between the scheduler and the OPERA. This API was not implemented in the first versions of OPERA, and was added for the integration with the rest of the OCARI stack.

The lastest API is based on duty cycle: OPERA will generate one or several packets per cycle, with an API which wraps around a finer grained API.

At the beginning of each cycle, the following function is called:

[2] because of the use in simulators: structures are required instead of global variables

[3] Note that as in current specification, none of the functions is expected to take significant time, they all have bounded execution time, which should be $O(n)$ in the number of nodes/neighbors with n bounded anyway - there is no MPR computation, no Dijkstra route calculation, no $O(n^2)$ table maintenance, except for the one moment when coloring (OSERENA) starts.

```
hipsens_bool opera_event_new_cycle(opera_state_t* state,
                                   byte unused);
```

This function is the main "scheduling" for OPERA. It is called periodically exactly once per OCARI cycle
- it returns whether OPERA wishes to have a packet buffer or not.
This function should be called only after `opera_init` was called.

If this function returns `HIPSENS_TRUE`, another function is called:

```
hipsens_bool opera_event_wakeup_with_buffer
(opera_state_t* state, transmit_buffer_t* transmit_buffer);
```

This function is called every time OPERA has indicated that it wishes to be called with a buffer (from return values of `opera_event_new_cycle`, or `opera_event_packet`). Its return value indicates whether `opera_event_wakeup_with_buffer` wishes to be called again with another transmit buffer ; in addition:
. if `payload_size` > 0, this indicates that the buffer should be sent and other fields are properly set (`next_hop_address`, `traffic_type`) ; and if `transmit_buffer` == `NULL`, this indicates that no buffer is available at time of the call.

Moreover, the other event handled by OPERA is packet reception:

```
hipsens_bool opera_event_packet_received
(opera_state_t* state, byte* packet_data, int packet_size, hipsens_u8 rssi);
```

The external caller should call this function every time a packet is received.

- 'rssi' corresponds to the link quality estimated for the received packet
- for OCARI, the 'rssi' implementation needs to be defined
- the return value indicates if OPERA wishes to send a packet immediatly (typically in reaction to the processing of the packet just received).

2.1.2 Finer-grained, internal, API

The previous API builds on a former, now internal, API.

The starting point, is that even if the OPERA submodules, require the scheduling of a callback at some point in the time, sometimes, the goal of the callback is to generate a control packet: however at the time of the awakening, it might be the case that no buffer is available at the lower layers to send a packet (e.g. MAC queues full). Then OPERA would have to wait later, in order to have an empty queue.

Before the change, OPERA was returning the time of the next wake-up given by the following function,

```
hipsens_time_t opera_get_next_event_time(opera_state_t* state);
```

Instead, now, OPERA also indicates whether or not a transmission buffer is required when it is awaken. However, even no transmission buffer is available, OPERA will not be awaken. OPERA might still wish to be awaken *without* buffer, if no buffer is available for a long time (e.g. handle expiration timers, ...).

So in order to be more complete, OPERA returns two values:

- the earliest time when OPERA should be awaken with a transmission buffer.
- the earliest time when OPERA should be awaken without transmission buffer.

This works with the following:

```
typedef struct s_hipsens_wakeup_condition {
  hipsens_time_t wakeup_time; /* without buffer */
  hipsens_time_t wakeup_time_buffer; /* with buffer */
} hipsens_wakeup_condition_t;

static void internal_opera_get_next_wakeup_condition
(opera_state_t* state, hipsens_wakeup_condition_t* condition)
```

Returns the next time the OPERA node should be woken up by the external caller (with the function `opera_handle_event_wake_up`) on one of the conditions (with buffer or without buffer for sending one packet).

2.2 Orchestrating module, OPERA

2.3 The external API offered by OPERA

The orchestrating module is represented by the `opera_state_t` which itself includes each of the 3 modules (`eond_state_t`, `eostc_state_t`, and `serena_state_t`), and a few other fields.

The orchestrating module offers three functions:

- A function "new cycle event": called once per cycle. Returns whether OPERA submodules want to send a packet.

- A function "handle event packet": handles an incoming event where a packet has arrived, and the module will dispatch it to the proper submodule EOND, EOSTC, or OSERENA
- A function "wake-up with transmission buffer": the caller wakes up the module OPERA; OPERA will consider which submodule actually should be woken up, and be offered the transmission buffer. This function also indicates whether the node wishes to transmit another packet.

2.4 OBSOLETE API: Using OPERA

The file `eosimul-simple.c` was a simple example of a use of OSERENA (written for this purpose). It is now **OBSOLETE**.

Precisely, OPERA was used as follows:

- The caller has a variable `opera_state_t`
- The variable is initialized by `opera_init` and OPERA is started with `opera_start`.

  A callback function is passed to OPERA, to send a packet.
- The main loop is as follows:
  - The caller gets at which point of time the OPERA module wants to awoken: this is given by `opera_get_next_event_time`.
  - The caller then waits for one of the following events:
    * the time at which OPERA wants to be awoken has arrived
    * – or – a packet has been received for OPERA (before the time of wake-up has arrived)

    In this case, the caller will use one of the functions `opera_handle_event_wake_up` or `opera_handle_event_packet` accordingly.
  - and the loop is repeated.

Notice that OPERA may generate packets during the calls `opera_handle_event_wake_up` or `opera_handle_event_packet`. Note also that if a packet is given for processing to OSERENA, the value returned by `opera_get_next_event_time` may change.

2.5 Internal interaction between sub-modules

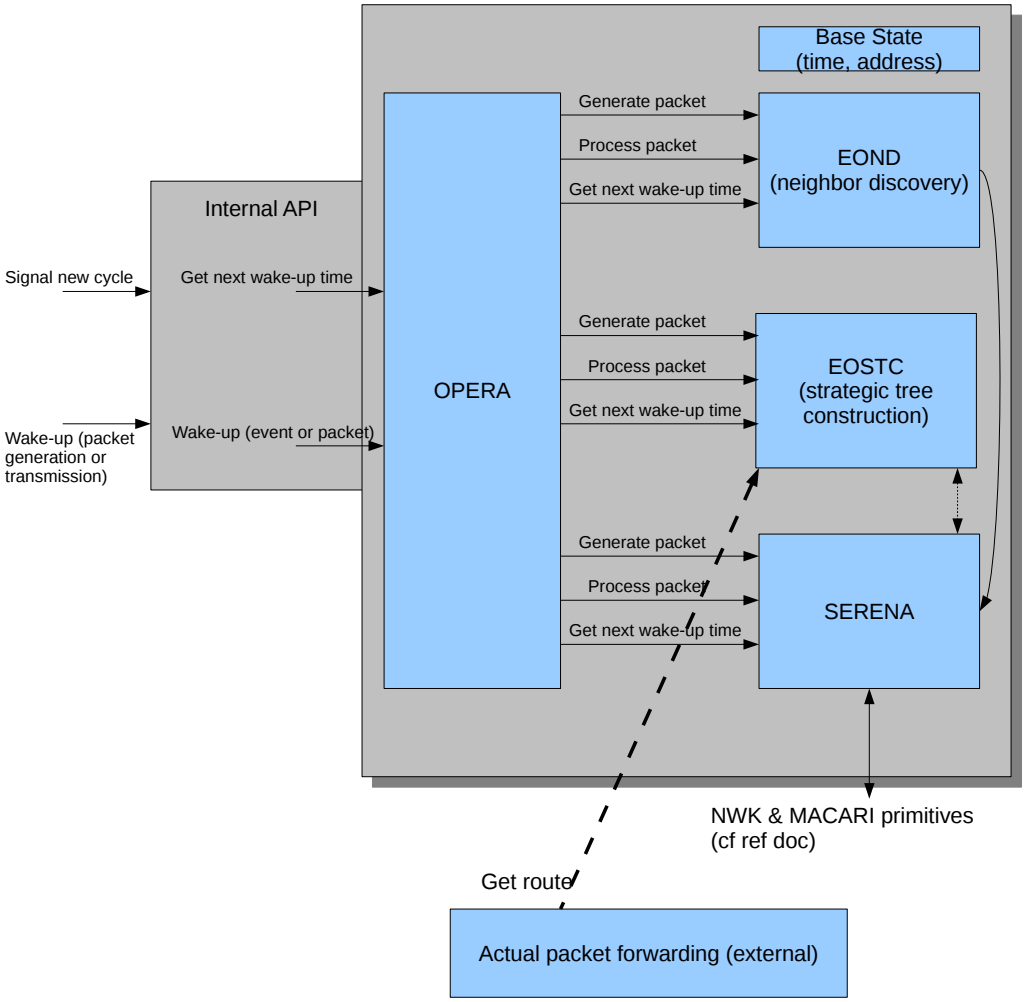
In addition, the module OPERA will include most of the "big picture" logic to allow proper management of the interaction of the different submodules that each them cannot handle individually. An example: any Hello generation should be also preceeded by neighbor table cleaning; neighbor table cleaning (in EOND) should be followed by tree table cleaning (in EOSTC); and finally tree table cleaning can result in Tree Status generation – also this implies a double packet generation which should be managed properly. Because none of the modules EOND, or EOSTC, has enough knowledge by itself to understand this complex interaction, the role of handling them properly is (will be) delegated to the orchestrating module.

The design is represented in figure 2.5

3 Files

This section described the organization of the files commited (this section has not been updated to the lastest API).

- `hipsens-config.h` defines the configuration and default parameters: for OCARI the content is mostly overidden by `hipsens-cc2530.h` where all necessary defines should be put (or in IAR compiler options), provided that `CC2530` is #defined
- `hipsens-macro.h` defines complicated macros (for debugging, logging, ...) depending on the environment (with/without printf, with/without fprintf). However because of the full-state-dump functionality in Python wrappers, logging and debugging macros are now mostly unused in practice (see below "about printf"). Among the macros there is also:
  - `WARN(base_state, ...)`
  - `FATAL(base_state, ...)`
  - `ASSERT(...)`
  - `PRINTF(...)` does nothing is `WITH_PRINTF` is not defined (default for OCARI)
  - `FPRINTF(...)` does nothing if `WITH_PRINTF` is not defined, does the same thing as `PRINTF` if `WITH_FILE_IO` is not defined, otherwise is equivalent to fprintf
  - the `STLOG/STWRITE/STLOGA/...` macros, but they include '`if (should_log)`' which is comes from a value currently #defined to `0` in `hipsens-cc2530.h`, hence the compiler is expected to remove them


Base State
(time, address)
Generate packet
Process packet
Get next wake-up time
EOND
(neighbor discovery)
Internal API
Signal new cycle
Get next wake-up time
OPERA
Generate packet
Process packet
Get next wake-up time
EOSTC
(strategic tree
construction)
Wake-up (packet
generation or
transmission)
Wake-up (event or packet)
Generate packet
Process packet
Get next wake-up time
SERENA
NWK & MACARI primitives
(cf ref doc)
Get route
Actual packet forwarding (external)

- `hipsens-base.h`/`hipsens-base.c` includes general functions/structures which were intially not specific to OCARI (and has grown...) It includes:
  - `base_state_t` = the base state for the node information shared in submodules, which currently includes the clock, the address of the node, the energy class.
  - time related macros:
    - `hipsens_time_t` = type for the representation of the time in OPERA
    - `HIPSENS_TIME_TO_SEC(...)`/... macros to convert seconds, milliseconds to `hipsens_time_t` (and back).
  - definitions of the integer types unsigned/signed 1 byte, 2 bytes, 4 bytes, and of boolean (such as `hipsens_u8`, `hipsens_s8`, ... `hipsens_bool`)
  - address related functions (copy, compare)
  - an unused attempt at a random generator
- `hipsens-opera.h`/`hipsens-opera.c` is the main module for access to EOLSR+OSERENA
  - although the modules EOND, EOSTC, OSERENA could be accessed directly, the goal of OPERA is to provide a more simple API which should be easier to understand and use (see `http://en.wikipedia.org/wiki/Facade_pattern`)
  - the status of the API is changing
- `hipsens-eond.h`/`hipsens-eond.c` includes all the EOLSR neighbor discovery (EOND) module.
- `hipsens-eostc.h`/`hipsens-eostc.c` includes all the EOLSR Strategic tree construction protocol (EOSTC). This module requires the EOND module. It is under construction.
- `hipsens-oserena.h`/`hipsens-oserena.c` includes all the OSERENA protocol. This module requires the EOND module. Later it will probably requires the EOSTC module, or some interaction between the two.
- `eosimul-simple.c` was a simple OBSOLETE example of the use of OPERA, it should not be included in the compilation

4 Implementation Specifics

4.1 Addresses

The code assumes that addresses are sequences of bytes of identical, fixed size, `ADDRESS_SIZE`. There is also the assumption of constant "null address".

4.2 Time management

The modules assume the existence of a general clock. The clock could be implemented by incrementing a counter on a periodic interrupt (as it is in Contiki for instance).

The modules also assume:

- a signed time type (code uses time differences)
- a special constant `zero_time` – which should be renamed `null_time` or `undefined_time` – stands for an "unspecified" time.

The clock wrap-around is currently not handled

4.3 About `printf`

The code can output its state as python datastructure (almost JSON).

However to use this functionnality `WITH_PRINTF` should be #defined When `WITH_PRINTF` is defined, the function printf is called ; where not available, cg-printf.c/cg-printf.h have been used but they are not added to the git repository right now.